24 | Spanner（三）：严格串行化的分布式系统

2021-11-26 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩
时长 22:58 大小 21.05M

你好，我是徐文浩。

Spanner 在设计时候的目标之一，就是需要保障外部一致性（external consistency）。而这个外部一致性，其实也就是我们之前说过的可线性化（Linearizability）。通过上节课的学习，现在我们已经知道了，这是一个非常有挑战的目标。其中**最大的挑战，在于不同服务器的时钟的差异。**

Spanner 采用了两个主要的策略来解决这个问题，第一个是通过原子钟和 GPS 时钟，尽可能地缩短服务器之间的时钟差异。另一个，则是在实现分布式事务的时候，把时钟差异考虑进去，实际的事务提交会进行一定的等待，确保数据写入一定是考虑了最大误差和参与者最快的时钟下，才会实际提交事务。

那么，今天我们这节课，就会深入来讲解 Spanner 的这个分布式事务具体是怎么实现的。并且在这个过程中，我们会深入探讨分布式事务的可串行化和可线性化两个概念的差异在哪里。在学完这节课之后，我希望你能够收获两点：

第一个，自然是 **Spanner 的分布式事务，具体是怎么实现的**。这个也有助于你了解最新一代的分布式数据库的实现，现在已经有不少新的数据库，都是师从 Spanner 的论文了。

第二个，是深入理解**分布式数据库的可串行化和可线性化概念的相关性和差别**。分布式数据库的主要挑战，就来自这两方面。

理解可串行化和可线性化

在之前讲解 Chubby、Megastore 和 Spanner 的过程中，我们既提过“可串行化”，也提过“可线性化”的概念。这两个概念，在分布式数据库里非常容易混淆，为了能够在讲解 Spanner 的事务实现的时候更加清晰，我先带你来梳理一下这个概念。

可串行化

可串行化是一个数据库事务隔离性的一个概念，它本身和分布式系统没有关系。它的定义是，一个事务的集合 S，可串行化是该集合中事务的一个满足全序关系的排列。换句话说，可串行化，相当于在外部看起来，这些事务是按照某一个顺序，一个一个执行下来的。

这个属性，主要是为了解决数据库里，在有并发事务存在的情况下，数据库应该做到什么样的隔离保障。还是回到我们之前的转账例子里：

事务一，张三原先账户里有 500 元，向李四转账 400 元。

事务二，李四原先账户里有 100 元，向王二转账 300 元。

这两个事务如果都执行成功，只可能事务一先执行完，再执行事务二。而不是在事务一执行的同时，李四的账户里钱已经增加了，但是张三的钱还没有扣除的时候，事务二就开始执行了。同样的，我们也支持先执行事务二再执行事务一，但是事务二会先失败，而事务一会执行成功。

但是，在可串行化的情况下，我们仍然完全有可能遇到事务一已经执行完成了，事务二再提交，但是执行失败的问题。事务二失败的原因是，它读到的仍然是事务一执行完成前的数据快照，认为李四的账户余额不足。

这种情况，在分布式的系统下很有可能发生。**事务一已经提交完成了，但是因为系统是分布式的，还没有同步到所有的副本。事务二只能读到旧的数据快照，但是为了确保事务的隔离性是可串行化的，我们不能让事务二提交完成。**

可串行化但是不可线性化的系统，很多时候是我们愿意容忍的，因为它至少确保了数据库里的数据的**正确性**。即使发生了前面事务失败的例子，我们仍然可以在应用操作层面，等待重试。

可线性化

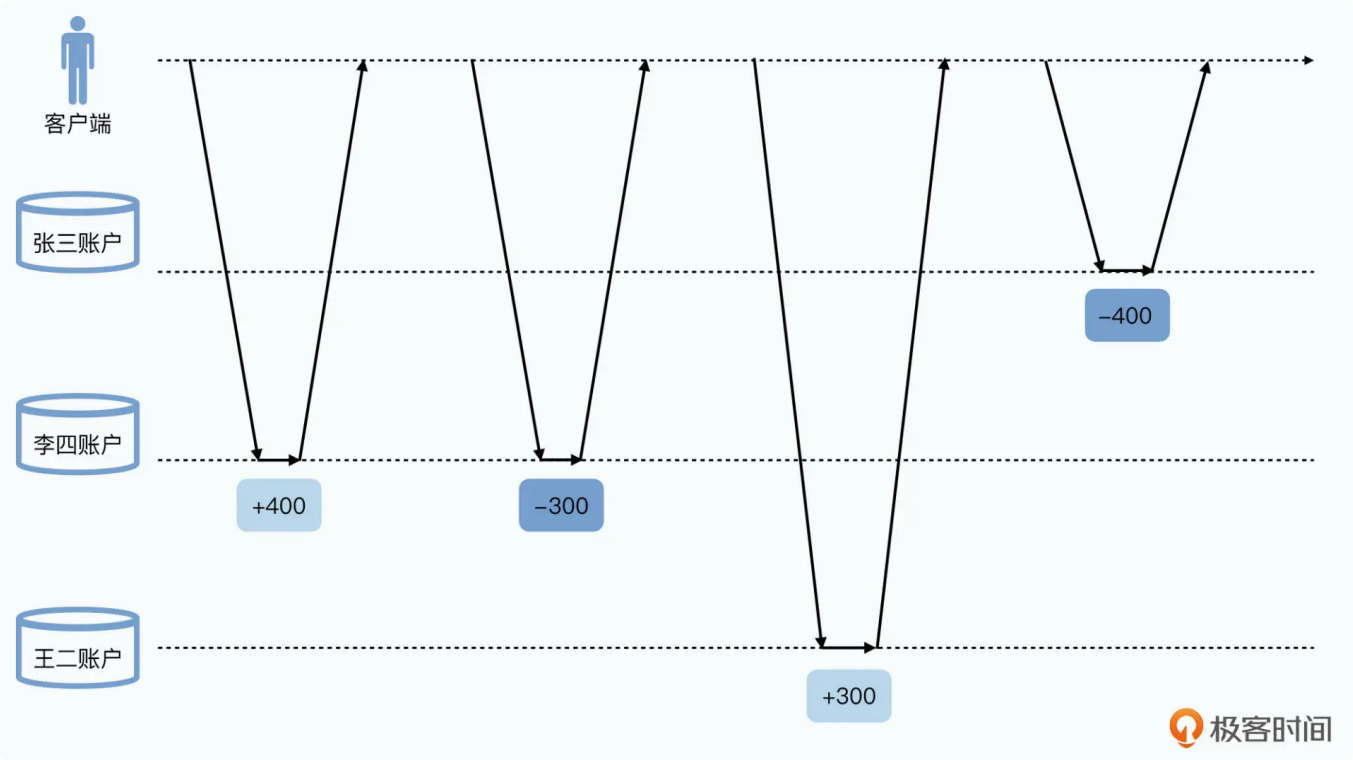
而可线性化是一个分布式系统中的概念。它的含义是，对单个对象上的操作，是“实时”的。也就是你对一个数据写入操作成功了，那么立刻去读取它，就会读到刚刚写入的值。

这个乍一听是理所当然的事情，其实无论是在单机环境下，还是分布式环境下都有可能被打破。在单机环境下，可能我们写入成功的数据还在 CPU Cache 里，没有同步到主内存，那么来自另外一个 CPU 的数据读取，可能就取不到这个值。

而在分布式环境下，这个问题尤其普遍。我们可能在上海的数据中心写入，1 秒之后在广州的数据中心去读的时候，数据还没有同步过来，CAP 理论里的一致性（Consistency）其实大部分情况下都是在探讨这个问题。其实多个 CPU，你也可以把它看成是单机环境里的一个“微观”的分布式系统。

同时，**可线性化完全可以和数据库事务的可串行化无关**。比如前面的转账例子里，如果没有事务，我们完全可以让所有的账户变动过程变成这样：

1. 李四的账户先多了 400 元；
2. 李四的账户里再少了 300 元；
3. 王二的账户里再多了 300 元；
4. 张三的账户里少了 400 元。



要注意，可线性化是针对单个对象的，而且和数据库的事务性不一定相关
这里的操作流程不满足事务性，特别是事务性中的“一致性”
但是因为系统中没有分布式的情况，所以仍然是可线性化的

从可线性化的角度来看，每个用户的**单个账户**的操作顺序都是满足了可线性化的。我们在第一步和第二步之间，只要读到李四的账户，是多了新转进去的 400 元就可以。但是因为没有事务支持，这样的操作顺序也满足不了我们实际的业务需求。

可线性化的本质，是需要我们在分布式系统上的操作符合常识。我们写入成功的数据，之后去读取，必须要能读取到刚刚写入的新数据，而不是有可能会读到一个旧版本的数据。即使写入和读取的动作，可能分别是从两个相聚几千公里的不同数据中心发起的。

如果做不到这一点，对应用开发人员来说，本来简单的逻辑会变得非常复杂，所有的业务逻辑，都需要去考虑因为网络延时或者时钟误差带来的紊乱。而我们本身设计分布式系统的一大目标，就是让应用开发人员不需要关心分布式的存在，那么不是“可线性化”的系统，显然做不到这一点。

不过，你需要注意，可线性化是针对**单个对象上**的操作。如果两个操作，或者是两个数据库事务，互相之间没有任何交集，那它们在系统里记录的事务的时间，是不需要保障先后顺序的。

比如，我们有这样两个事务：

事务一，张三向李四转账 400 元。

事务二，王二向赵大转账 200 元。

这两个数据库事务之间，什么顺序都可以，因为两个事务之间没有共同操作的对象。而只有我们再有一个：

事务三，李四向王二转账 300 元。

在事务三出现的情况下，我们才需要对事务一、二、三明确一个排列顺序。因为事务三操作的数据，既和事务一有交集，也和事务二有交集。所以事务一和二在这个情况下，是通过事务三关联起来，才会有明确的顺序。

严格串行化

而在 Spanner 这个数据库里，我们需要同时保障可串行化和可线性化。而可串行化和可线性化的组合，在分布式数据库领域里，被称之为**严格串行化**（Strict Serializability）。也就是在外部看来，事务是串行执行的，并且一旦一个事务执行成功，我们立刻去读取数据的话，无论是从分布式系统的哪个节点发起，我们一定能够读取到刚才的最新数据。

这个体验，其实就跟我们日常生活里的常识和体会一模一样了。你转账成功之后，告诉你远在千里之外的亲友，他去查看银行卡，就一定能看到银行卡里钱已经转进来了，而不是有可能需要等待网络数据的同步。

我们的数据库，不仅因为支持可串行化的事务隔离性，在内部看起来是一致的，对于所有外部使用数据库的人来说，也是一致并且正确的。这也是为什么 Spanner 说它自己是一个“外部一致”的数据库。

Spanner 事务的具体实现

因为要支持严格串行化，解决时钟误差问题就对 Spanner 尤为重要。所以 Spanner 里的时间，并不是给出了一个简单的数字，来代表某一个时间点。而是通过一组叫做 **TrueTime API** 的接口，提供了一个有误差范围的“时间”概念。

TrueTime API

方法	返回值
TT.now()	*TTInterval: [earliest, latest]*
TT.after(t)	true 如果 *t* 这个时间点确定已经过了
TT.before(t)	true 如果 *t* 这个时间点确定还没有到

极客时间

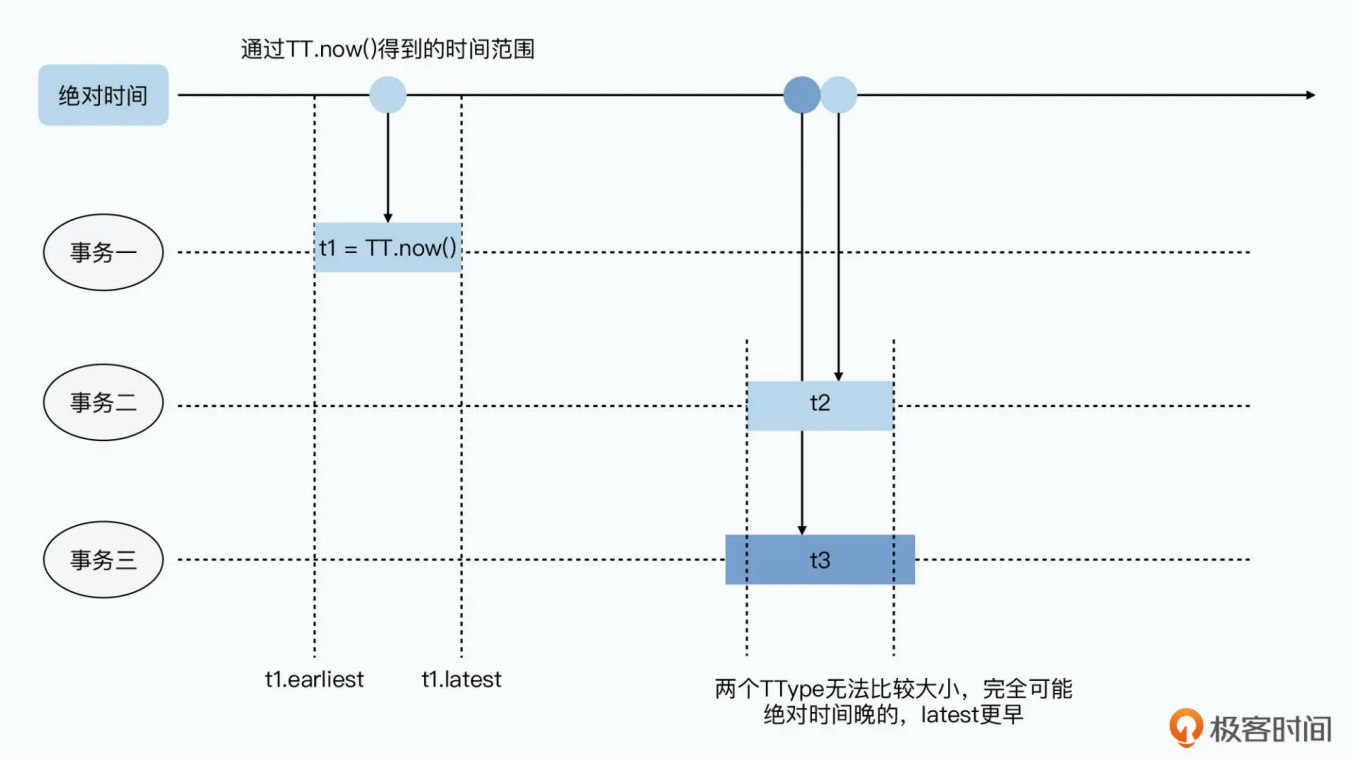
论文中的表1，TrueTime API，其中的参数 t 是 TTstamp 类型

首先，Spanner 假设有一个绝对准确的时钟，这个时钟的每一个时间点，就是一个**绝对时间**。Spanner 的论文里，对于任何一个事件的发生，都会有一个绝对的时间点，这个时间点，在论文里用 $t_{abs}(e)$ 来表示。其中的 e 代表 event，就是某一个事件，而 t_{abs} 则是绝对（absolute）时间（time）的意思。

我们任何一个服务器去获取当前的时间，是拿不到绝对时间的，而只能拿到一个 [earliest, latest] 这样的时间范围。同时，Spanner 也简单封装了 TT.after 和 TT.before 这两个接口，可以拿来比较服务器获取到的时间和绝对时间 $t_{abs}(e_{now})$ 的先后。

对于 TrueTime API 来说，它保障的是：

> *tt = TT.now()*
> *tt.ealierst* < $t_{abs}(e_{now})$ < *tt.latest*



TrueTime API拿到的是一个包含误差的绝对时间范围

Spanner 为了要保障可线性化，本质上就是要确保任何一个在**绝对时间 t** 发生的数据读取，一定只能看到在 t 这个时间节点之前已经提交成功的数据写入，之后的数据写入它是看不到的。并且，只要是在绝对时间 t 之前已经提交成功的事务写入，它也一定要能看到，而不能有些看不到。

那么，Spanner 是怎么通过自己的系统架构和这里的 TrueAPI，做到这一点的呢？我们一起接着往下看。

Spanner 支持的读写类型

我们先来看看 Spanner 支持的数据读写的事务类型：

操作	关于时间戳的讨论	并发控制	必须的副本
读写事务	论文 4.1.2 章节	悲观锁	Leader
快照事务	论文 4.1.4 章节	无锁	通过Leader获得时间戳；通过任意副本去读，参看论文4.1.3
快照读，由客户端选择时间戳	--	无锁	任意副本，参看论文4.1.3
快照读，由客户端选择时间上届（upper bound）	论文 4.1.3 章节	无锁	任意副本，参看论文4.1.3

极客时间

Spanner 一共支持 3 种操作，分别是**读写事务**、**快照读事务**、以及**普通的快照读**。

其中，**只有读写事务可以修改数据库里的数据**，读写事务需要使用悲观锁，也就是对应的数据行一旦被锁，其他读写事务，是不能并发修改对应数据的。而所有的读数据都是无锁的，也就是无论是否有其他并发的读请求，或者有读写事务在进行，都不会影响数据的读取。

快照读事务，和普通的快照读，**唯一的差异就在于这个快照的“时间戳”由谁来提供。**如果是快照读事务，那么这个“时间戳”是由 Spanner 系统来提供的。而普通的快照读，则是由应用指定时间，Spanner 里支持两种指定时间的方式，一个是客户端选择一个时间戳，另一种则是指定一个时间戳的上届（upper bound），让 Spanner 在这个上届之下，为客户端选择一个时间戳。

注意，我们这里说的时间戳，都是指一个**绝对时间**戳，而不是某一台服务器本地时钟的时间戳。

读写事务

我们先来看一下 Spanner 的读写事务是怎么进行的。我们在上节课看过，**Spanner 的数据库事务面临的第一个挑战，就是一个事务写入的时间戳，到底应该是什么。**Spanner 的读写事务是这样的：

首先，我们先把每一个 Paxos 组（Paxos Group）看成是一个单元，一个 Paxos 组内，数据就是完全通过 Paxos 协议来做到同步复制的，我们先不用去关心它内部的细节，可以先把它当成是一台服务器来看。

我们的分布式事务，其实是多个 Paxos 组参与的二阶段事务提交。其中，我们会选定一个 Paxos 组是协调者，其余 Paxos 组都是参与者。

那么整个读写过程会是这样的：

第一步，客户端写入数据库的请求，会先到达分布式事务中的协调者。而且在 Spanner 里，这个协调者本身也是一个事务的参与者。

第二步，和一般的两阶段事务一样，协调者会向参与者的 Paxos 的 Leader 发起一个**提交请求**，也就是 Prepare 请求。这个时候，参与者会为要更新的数据获取一个写锁。参与者会为事务生成一个 Prepare 请求的时间戳，并且向 Paxos 里，写入一条 Prepare 请求的记录。这个时间戳，会比这个 Paxos 组之前分配给所有事务的时间戳都要晚。

也就是说，这个 Paxos 组，在分配时间戳的时候会保障**单调递增**，确保不会出现，后出现的请求时间戳早的情况。

第三步，协调者本身不会给自己发 Prepare 请求，它会等待所有的参与者，返回事务是否可以提交的应答。这些应答中，参与者会把分配给 Prepare 请求的时间戳，一并返回给协调者。这个时候，协调者会为这次事务生成一个**事务提交的时间戳**。这个时间戳，需要满足三个要求：

首先，它要相等或者晚于其他参与者返回的 Prepare 请求被记录下来的时间戳。这个非常直接，从时序关系上，我们不能让第二阶段的事务提交的时间比第一阶段的事务准备的时间还要早。

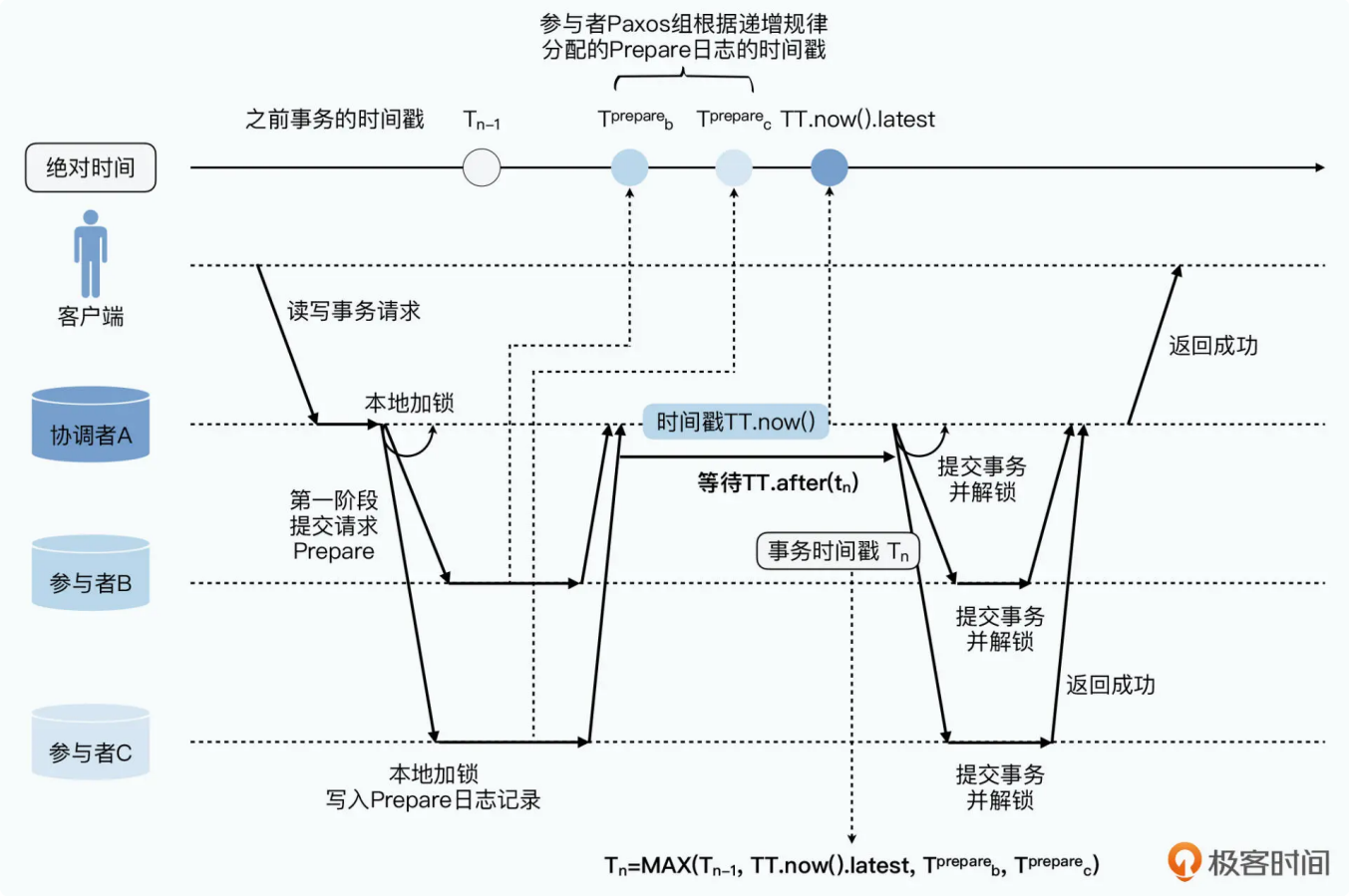
其次，它要比当前的 Leader 分配给之前事务的时间戳要晚。这个也非常直接，我们不能让先提交的事务有一个更晚的时间戳。

最后，它需要晚于 TT.now().latest，也就是协调者自身当前的时间戳，并且这个时间戳需要把时钟误差考虑进去。因为我们实际事务提交的时间，是比我们确定要提交事务的时间晚的。

第四步，在有了这个时间戳之后，协调者会进行一个有条件的等待。它会等到 TT.after(s)，也就是在考虑误差的情况下，绝对时间一定已经晚于这个选择的事务提交时间之后，再提交事务。

提交事务的时候，作为一个参与者，它会在本地应用这个事务。作为协调者，它会向其他的参与者发起请求，让它们也提交事务。所有参与者写入的数据都有相同的时间戳，并且在事务应用完成之后，参与者会把对应的写锁释放掉。

第五步，在整个事务提交完成之后，协调者会返回结果给到客户端。



Spanner为事务生成的时间戳，需要考虑所有参与者的时钟，并且任何一个Paxos组生成的时间戳都必须是递增的

在这个机制下，当有两个并发的事务，需要写同一条记录的时候，我们能够确保它们写入事务的时间戳，是有明确的先后关系的。我们还是以上节课的转账例子来举例：

事务	时钟	用户	数据副本的Leader所在的Zone	账户余额	事务
事务一	0	张三	A	1000	转账500给李四
事务二	-2	李四	B	500	转账300给王二
事务三	-2	王二	C	700	转账600给张三

极客时间

我们之前遇到的时间戳问题，在于事务二后执行的时候，因为 B 和 C 的时钟比 A 要快。所以，事务二执行的时候，提交的时间戳可能早于事务一。

但是在现在这个机制下，我们看到：

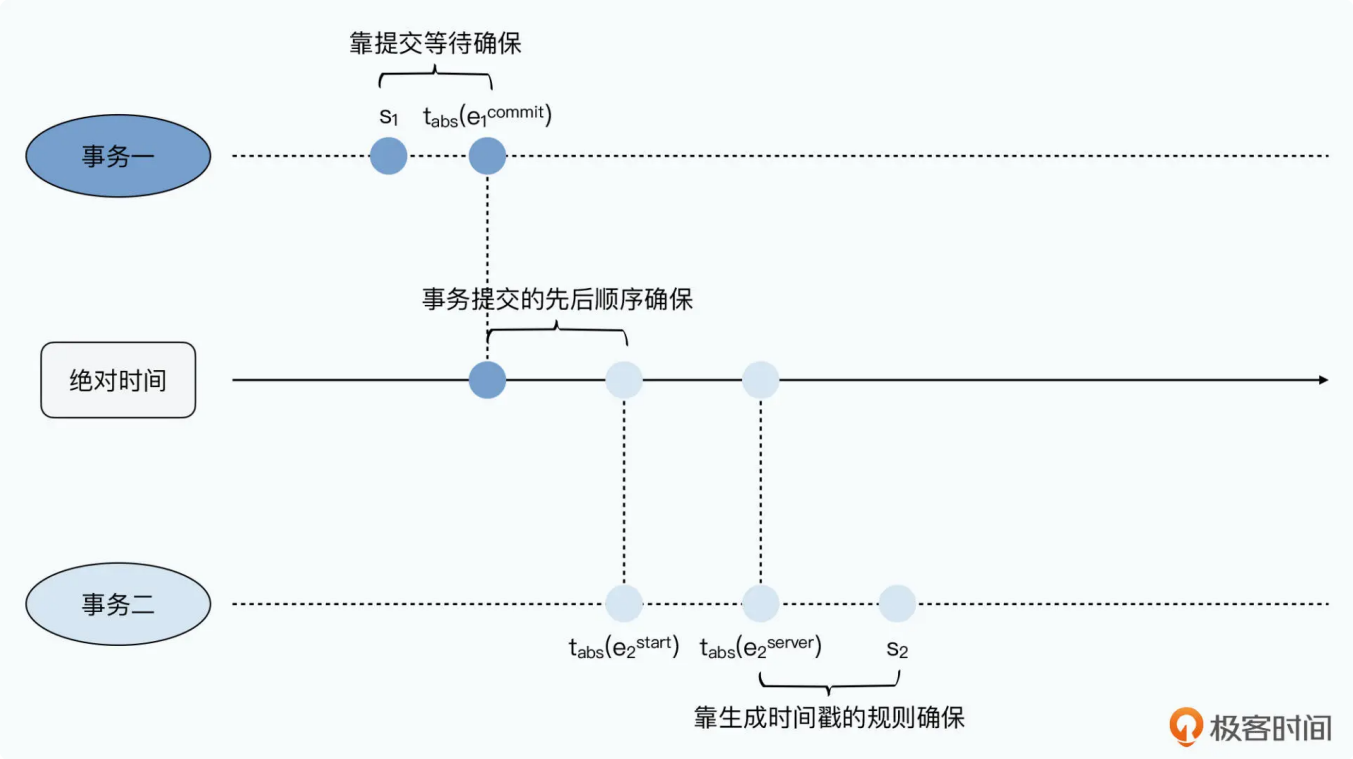
> 事务一，会使用 A 和 B 当中，较慢的时钟，来生成事务提交的时间戳。
>
> 事务提交之后，B 这里的 Paxos 里，最新分配的事务时间戳，需要晚于刚才已经提交的事务。所以事务二生成的事务提交的时间戳，一定会晚于事务一。

在 Spanner 的论文中，有一段很简短的证明，告诉我们如果两个事务牵涉到的 Paxos 组，是有交集的情况下，那么后提交的事务的时间戳，一定会晚于新提交的事务。我们可以在这里一起看一下。

1. 首先，$s_1 < t_{abs}(e_1^{commit})$，也就是系统为事务一生成的时间戳，一定早于事务实际提交的绝对时间。这一点，Spanner 在事务提交的时候，会进行**提交等待（commit wait）**，来确保这一点。
2. 其次，$t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start})$，也就是事务二的开始时间，晚于事务一的提交时间。这个是显然的，首先，我们的事务本身有提交的先后顺序。其次，因为 Spanner 的事务采用了悲观锁，两个读写事务如果需要写入相同的数据记录，后一个事务需要等到前一个事务，把对应的锁释放出来才能开始。所以，显然，后一个事务开始的绝对时间，是晚于前一个事务提交的绝对时间的。
3. $t_{abs}(e_2^{start}) \leq t_{abs}(e_2^{server})$，也就是事务二开始的绝对时间，是早于事务二决定提交事务的绝对时间的。这个也顺理成章，事务总是要先开始，才能决定提交。

4. 最后是 $t_{abs}(e_2^{server}) \leq s_2$。也就是事务二选取的时间戳，是晚于事务提交的绝对时间的。这个，是因为我们在选取时间戳的时候，是在决定提交之后，选取本地 TrueTime 的时间范围内最晚的，只要我们的误差可信，那么这个时间戳一定晚于决定提交那一刻的绝对时间。

然后根据大小比较的传递关系，我们就自然有 $s_1 < s_2$，也就是事务二记录的提交时间，是晚于事务一的。



通过提交等待，以及生成时间戳的规则，我们可以确保后提交的事务，必然时间戳是晚于先提交的事务的

可以看到，**在这个读写事务的机制下，我们的事务有可能需要等待一段时间，而且这个时间是和我们的时钟误差相关的。**时钟误差越小，我们需要等待的时间越短。这也是为什么 Google 需要在数据中心里，采用原子钟和 GPS 时钟，尽可能缩短不同服务器的时钟差异的核心原因。

快照事务和快照读取

既然我们已经能够保障，Spanner 的事务写入的时候的时间戳没有问题。那么，下一个挑战就来自于数据库读了。**因为 Spanner 允许从任何一个副本读取数据，所以我们就会面临另一个挑战，就是 Paxos 下，我们并不能保障每一个副本都拥有最新的数据。**

因为数据的写入只需要多数通过就好了，我们完全可能因为网络分区、数据延时等问题，访问了一个没有最新版本数据的副本。那么这个时候，我们该怎么办呢？

Spanner 是这样解决这个问题的。Spanner 里的数据库读取，都会带上一个时间戳，如果客户端自己没有指定，那么系统会为它分配一个时间戳，这个时间戳我们记为 s_{read}，这个值是由对应记录的 Paxos 组的 Leader 生成的。**最简单直接的方式**，就是使用 Leader 的 TT.now().latest，也就是考虑时间误差下，当前可能的最晚时间。

而当一个读取数据的请求到达某一个数据副本的时候，这个副本需要回答一个问题，就是它本地的最新数据是什么版本的呢？它是可以直接返回本地的数据，还是要让这个读取请求，等待到它把数据同步完成？

所以，每个副本会在本地维护一个称之为 t_{safe} 的值。只要请求带来的 s_{read} 比这个值小，那就说明本地副本的数据足够新，可以返回 s_{read} 这个时间节点下最新的数据快照。这个 t_{safe} 是这样取值的：

$$t_{safe} = min(t_{safe}^{paxos}, t_{safe}^{TM})$$

首先，每个副本会维护一个当前已经被应用了最大 Paxos 写入的时间戳 t_{safe}^{paxos}。这个时间戳，也就是已经提交并且更新到数据库里的最新的事务的时间戳。

其次，这个副本会维护另一个时间戳 t_{safe}^{TM}。当前副本所在的 Paxos 组里，不仅有已经提交完成了的事务，可能还会有正在进行中的事务。对于正在进行中的事务，我们并不知道事务最终提交的时间戳会是多少。但是，我们可以拿到所有这些事务，在 Prepare 阶段写入日志记录的时间戳，实际事务提交的时间 $s_{i,g}^{prepare}$，一定晚于这个时间戳。我们只要在所有这些进行中的事务的时间戳里面，选取最小的一个减去 1，那么我们就会得到一个时间戳 $t_{safe}^{TM} = min_i(s_{i,g}^{prepare}) - 1$。接下来不会有事务的提交时间，是早于这个时间戳的。

这样一来，我们就只要在 t_{safe}^{paxos} 和 t_{safe}^{TM} 中，选取较小的那一个时间戳。在这个时间节点之前的所有数据，当前副本都已经同步完成了。如果我们的 s_{read} 比这个时间早，那么我们就可以返回数据，否则我们就要等待 t_{safe} 这个时间大于 s_{read} 之后，才能返回查询结果。

不过这里问题来了，如果 Paxos 组里没有数据更新，我们的 t_{safe} 岂不是一直不会更新，我们的读请求就需要一直死等在那里？Spanner 的系统并没有那么傻，它通过这样一个机制来解决这个问题。

首先，是每一个 Paxos 组的 Leader，都会维护一个映射关系，叫做*MinNextTS(n)*。

这里的 n，指的是第 n 次 Paxos 写入。看着这个映射函数的命名，相信你不难猜到它是什么意思。Min 代表最小的，Next 代表下一次，而 TS 自然是 TimeStamp 的缩写也就是时间戳。这个映射函数是给到我们，下一次 Paxos 写入，能够分配到的最小的时间戳。

所以每一次 Paxos 数据写入之后，我们都会生成这个时间戳，然后 Paxos 的 Leader 会把我们的 t_{safe}^{paxos} 置为*MinNextTS(n)-1*，也就是下一次数据可以写入之前的时间节点。这样，会使得 t_{safe}^{paxos} 尽可能晚，使得它不容易小于 s_{read}，导致数据读请求需要等待。

此外，Paxos 的 Leader 即使在没有数据写入的情况下，也会默认每 8 秒钟更新一次*MinNextTS(n)*，既然时间已经往后推移了，自然 t_{safe}^{paxos} 也会对应向后更新。

所以，即使在最坏的情况下，我们的 Paxos 组，也能够至少返回 8 秒前时间戳的数据读请求。

其实，在没有进行中的事务的情况下，我们的 Leader 分配 s_{read} 可以更简单一些。它不会去分配 TT.now().latest，而是直接会拿到当前 Paxos 组最大写入的时间戳，直接用这个时间戳去获取数据就好了。否则，我们的请求多多少少都会阻塞等待，或者需要客户端去从其他副本读取，这样读取性能会变得差上不少。

小结

好了，到这里，Spanner 的论文我们就算是正式解读完了。在这节课里，我们先是回顾了“可串行化”和“可线性化”的概念。前者，想要解决的是数据库里的隔离性问题，后者则是想要解决分布式系统里的一致性问题。而 Spanner 则两个都要，也就是需要**实现“严格串行化”**。

Spanner 实现严格串行化的方式，是引入了 **TrueTime API**。通过这个 API，任何一个服务器拿到的本地时间，不再是一个精确的时间戳，而是一个**绝对时间**的范围。而在 Spanner 的数据写入的过程中，选择的时间戳，也是要考虑所有的参与者的时钟，并在其

中选择最晚的那一个。并且在提交的时候，也要考虑误差，确保事务写入的绝对时间，是晚于系统生成的时间戳的。

而在数据读取的时候，我们需要考虑对应的副本，不一定已经同步了最新的数据。所以所有的数据读取，都会带上一个**时间戳**。而每一个数据副本，也需要为一个本地的 Paxos 最新写入数据的时间戳。同时，读取数据也还要考虑当前是否有事务进行到一半。

在考虑了所有已经同步的数据和在进行中的事务中，选取出最早的时间戳，就是当前副本确定拥有的最新的更新了。通过对比本地副本的这个时间戳和读请求的时间戳，我们就能知道是否可以返回数据，还是需要等待本地副本进一步同步完数据才行。

Spanner 的论文是非常重要的，对于分布式数据库系统也有很强的里程碑意义。它告诉全世界，**一个强一致、高性能、全球分布式的数据库是完全可行的**。在 Spanner 的论文之后，像 TiDB、CockroachDB 这些开源的数据库，也都跟随 Spanner 的脚步，以它为蓝本去实现支持完整 ACID 的分布式关系型数据库了。

推荐阅读

可串行化和可线性化是数据库系统，以及分布式系统中一个非常重要的话题。我推荐你可以去读一读这一篇关于一致性模型的博客，对于可线性化和可串行化，以及每一个概念都有详细的解说。对应的各个概念的脑图，你也都可以点击进去，看看每个概念的详细解释。

思考题

在讲解 Spanner 的数据写入的时候，我们是直接把一个 Paxos 组看成是一个“单机”来处理的。一个 Paxos 组，需要自己保障生成出来的时间戳的单调递增。但是，实际上 Paxos 组也是有多个机器组成的分布式集群，而且这些机器之间，也是有时钟误差的。而且 Paxos 组里的 Leader 只拥有一个 0~10 秒的租约，这意味着 Leader 本身，会切换到另一台时钟不同的节点上去。

那么在读完论文之后，你能说说 Spanner 是如何处理和解决这里的时钟差异的吗？

欢迎在留言区分享你的答案和思考过程，也欢迎你把今天的内容，分享给更多的朋友，一起交流讨论。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 23 | Spanner（二）：时间的悖论

下一篇 复习课（一）| The Google File System

精选留言 (2)

写留言

在路上

2021-12-02

徐老师好，论文第4.2.2节提到，如果只读事务只涉及一个paxos group，读操作的时间戳选择Leader的LastTS()，如果涉及多个paxos group，读操作的时间戳选择本地的TT.now().latest()，这个时间可能会超过t_paxos_safe，需要等待safe time的推进。在没有新的事务发生时，如果依靠每8秒钟更新一次的MinNextTS(n)推进safe time，我觉得读操作太慢

了。我在Life of Cloud Spanner Reads & Writes中读到多节点的Strong Read，副本不知...

展开

在路上

2021-11-30

徐老师好，在Spanner论文的最后一页，描述了Paxos Leader-Lease Management，证明了Leader切换后时间戳的单调递增性。Leader选举需要多数Paxos成员的支持，所以一定存在一个Paxos成员同时参与了前后两轮选举，假设这个成员为r0，任何Paxos成员参与Leader选举之后，10秒之内不会再参与选举，Leader租期的结束时间由投支持票的Paxos成员的投票时间决定，选择最小的投票时间加上10秒。如果r0的投票时间是最小时间，那么下...

展开